Simulative Analysis of the RapidIO Embedded Interconnect Architecture for Real-Time, Network-Intensive Applications

David Bueno, Adam Leko, Chris Conger, Ian Troxel, and Alan D. George
{bueno, leko, conger, troxel, george}@hcs.ufl.edu

High-performance Computing and Simulation (HCS) Research Lab, Department of Electrical and Computer Engineering, University of Florida, P.O. Box 116200, Gainesville, Florida 32611, USA

Abstract—**RapidIO is an emerging standard for switched interconnection of processors and boards in embedded systems. In this paper, we use discrete-event simulation to evaluate and prototype RapidIO-based systems with respect to their performance in an environment targeted towards space-based radar applications. This application class makes an ideal test case for RapidIO feasibility study due to high system throughput requirements and real-time processing constraints. Our results show that a baseline RapidIO system is well suited to space-based radar, providing significant improvements over typical bus-based architectures. Our results also show that extensions to the RapidIO protocol such as cut-through routing and transmitter-controlled flow-control would provide minimal performance improvements for the applications under study.**

Index Terms—**RapidIO, space-based radar, ground-moving target indicator, embedded signal processing**

I. Introduction

RapidIO is a high-performance, packet-switched interconnect technology targeted towards processor-to-processor and board-to-board interconnection in embedded systems. In this paper, we use simulation to evaluate several systems built around RapidIO interconnects. Specifically, we consider architectures designed to support one variant of a space-based radar (SBR) algorithm known as Ground Moving Target Indicator (GMTI) by exploring various architecture options for the RapidIO interconnect, as well as three different parallel partitionings of the GMTI algorithm.

Today's state-of-the-art satellite signal-processing systems are typically built around a custom interconnect comprised of a menagerie of point-to-point and bus-based protocols. Custom interconnects often provide a high-performance platform for a particular system but at a cost of flexibility when applied to other systems and applications. A recent trend toward commercial-off-the-shelf (COTS) components has sought to improve system flexibility by designing a range of systems built upon inexpensive and interoperable, standard components. At the core of most satellite systems proposed today is a COTS bus interconnect such as the VME bus or more recently the PCI bus. Compact PCI (cPCI) is an adaptation of the PCI specification designed to provide a more robust mechanical form factor than traditional PCI, making it ideal for satellite systems. However, Compact PCI is subject to all of the fundamental limitations of bus-based architectures such as potential for contention and limited available bandwidth. While a 64-bit, 33 MHz cPCI bus can provide roughly two gigabits per second of throughput, a single 8-bit, double data rate (DDR) 250 MHz RapidIO endpoint can provide almost double this throughput by sampling data on both edges of the clock. In addition, as RapidIO is a switched network, it is much more scalable than a bus-based network, and even modest systems will provide tens of gigabits per second of aggregate throughput with many non-blocking links. This increased throughput provides the necessary network capabilities to perform real-time, network-intensive applications such as GMTI on flight systems.

GMTI is an algorithm composed of several subtasks, and is used to detect and track movement of ground targets from air or space. Algorithms used in space-based GMTI are very similar to those used in air-based GMTI. However, the input data sets for GMTI systems based in space can be much larger than those associated with air-based systems (due to line of sight) yet space-based platforms typically have more stringent operating tolerances. Therefore, the design of architectures related to space-based GMTI may be much different than those of air-based GMTI. GMTI relies heavily on signal processing, and the work performed inside each subtask does not require any processor-to-processor communication or synchronization. However, depending upon how the algorithm is decomposed, varying amounts of communication may be needed between each subtask. After the incoming data set is processed, a set of detection results is produced that is generally a few orders of magnitude smaller than the incoming data set. Due to the communication delays associated with space-to-ground downlinks and the large size of data associated with space-based GMTI, it is advantageous to process the incoming radar data onboard the satellite in a real-time manner. This real-time processing requirement has important implications, which will be discussed later in this paper.

The remainder of this paper is organized in the following manner. Section 2 describes previous work related to both RapidIO and GMTI, and Section 3 introduces the reader to important concepts and background information for RapidIO and GMTI. Section 4 describes the GMTI algorithms used for our experiments, along with the conditions and environment used in our simulations. Section 5 presents the simulation results, and Section 6 provides an analysis of these results. Finally, Section 7 provides conclusions and directions for future work.

We wish to thank Honeywell Space Systems in Clearwater, FL for their sponsorship and technical guidance in support of this research.

II. Related Work

As mentioned previously, RapidIO is a fairly new technology, and thus does not have a vast body of research relative to other interconnects such as InfiniBand [1] or the Scalable Coherent Interconnect (SCI) [2]. The main reference for RapidIO is the set of formal specifications, published by the RapidIO Trade Organization (RTO) [3-8]. These specifications define the RapidIO protocol, and are extremely detailed documents. Motorola is a key company driving the development of RapidIO, and has published many whitepapers and conference presentations regarding the protocol and associated issues.

Work on GMTI has received attention over recent years [9-10]. Space-time adaptive processing (STAP) in particular, one of the main components of any GMTI-based system, has also received much attention [11-12]. One interesting set of related papers on GMTI comes from work at the Air Force Research Labs in Rome, NY. In late May and June of 1996, a series of flights were performed in which a ruggedized version of Intel's Paragon supercomputer system was flown aboard a surveillance airplane and real-time processing of STAP was performed [13-14]. Partitioning of processing was performed via staggering of incoming data between processing nodes of the Paragon. One drawback of this partitioning method was that while it provided the necessary throughput to perform real-time STAP processing, each processing result had a latency of approximately 10 times the incoming data rate. A group of researchers at Northwestern University attempted to improve this situation by examining a parallel-pipelined version of the STAP algorithm used, and the results are presented in [15-16]. Our pipelined version of the GMTI algorithm is based on their work. Of course, one of the major challenges for space-based systems is getting the processing power of a Paragon supercomputer (or more) into a self-contained system while meeting the strict size, weight, power, and radiation requirements imposed by the space environment.

III. Background

In this section, we introduce the reader to basic information about RapidIO and many of the design options that are available for RapidIO implementations. In addition, we give an introduction to concepts and terms involved with GMTI and STAP algorithms.

A. *RapidIO*

RapidIO is an open standard for a high-bandwidth, packet-switched interconnect that supports data rates up to approximately 60 Gbps. RapidIO is available in both parallel and serial versions, with the parallel versions of RapidIO supporting much higher data rates. RapidIO uses the Low-Voltage Differential Signaling (LVDS) technique to minimize power usage at high clock speeds, and therefore is appropriate for use in embedded systems. RapidIO, like other packet-switched networks, employs Multistage Interconnection Networks (MINs) to allow communication between arbitrary devices with RapidIO endpoints.

One defining characteristic of the RapidIO protocol is that it is designed with simplicity in mind so that it should not take up an exorbitant amount of real estate on embedded devices. The RapidIO protocol is a layered protocol comprised of three layers: the logical layer, the transport layer, and the physical layer. The logical layer supports at least three different variants: a version suited to simple I/O operations, a message-passing version, and a version supporting cache-coherent shared memory. For this project, the message-passing version of the logical layer provides the least amount of overhead for moving large amounts of data throughout the RapidIO network.

The message-passing layer provides applications a traditional message-passing interface with mailbox-style delivery. The message-passing layer supports 26 message priorities and handles segmenting messages of up to 4096 bytes into packets (each RapidIO packet has a maximum payload size of 256 bytes). The physical layer only supports 4 priority levels, so the message-passing layer maps its 26 priority levels onto the 4 priority levels of the physical interface. Each message-passing transaction is composed of two events: a request and a response. For the purposes of our project, the response notifies the application layer that a specific RapidIO packet has been delivered to the destination.

The RapidIO physical layer provides several different options based on the level of performance that is needed. The physical layer is source synchronous and supports several different clock speeds, including 250 MHz, 500 MHz, and 1 GHz, and comes in serial and parallel versions [4]. The parallel version is geared towards applications that need very high amounts of bandwidth over short distances and comes in widths of 8 or 16 bits. The serial version of RapidIO is geared towards forming backplanes over somewhat longer distances than the parallel version. Both the parallel and serial versions are bidirectional, and data is sampled on both sides of the clock edge (DDR). The available effective data rates for RapidIO span from 1 Gbps for the serial versions of RapidIO up to 60 Gbps for the 16-bit parallel versions of RapidIO. For low-power environments associated with space-based radar systems, the 16-bit parallel version of RapidIO at lower clock speeds provides a good opportunity to keep data transfer rates high while minimizing power consumption.

In RapidIO, there are two main types of packets: regular packets and special control symbols. Regular packets contain data payload and other application-specific information associated with the logical and transport layers. Control symbols exist in order to ensure the correct operation of the physical layer and support flow control and link maintenance. In the parallel versions of RapidIO, control symbols are embedded inside packets in the LVDS signal. This interesting approach allows control symbols to be given the highest priority, although one can imagine that embedding control symbols complicates the hardware needed for RapidIO physical layer implementations.

Error detection is supported directly in the physical layer through the use of cyclic redundancy checks (CRCs) in regular packets and inverted bitwise replication of control symbols. The physical layer specification has protocols designed for error detection and recovery, and error recovery is accomplished through the retransmission of damaged packets.

The RapidIO specification for the physical layer provides two options for flow-control methods: transmitter-controlled flow control and receiver-controlled flow control. Both flow-control methods work with a "Go-Back-N" sliding window protocol for sending packets. In RapidIO, flow control is performed between each pair of electrically linked RapidIO devices. The specification requires receiver-controlled flow control to be available in all implementations of RapidIO devices, while transmitter-controlled flow control may be implemented optionally. In receiver-controlled flow control, a receiver of a packet notifies the sender of whether that packet has been accepted or rejected via control symbols. The sender makes no assumptions about the amount of buffer space available for incoming packets and relies on negative acknowledgements and retransmissions to ensure correct packet delivery. In transmitter-controlled flow control, the receiver notifies its link partner how much buffer space it has available by embedding this information into control symbols. The sender will never send more packets than the receiver has space. In this manner, the flow-control method will work more efficiently.

Since the physical layer supports four priority levels, the RapidIO specification suggests allocating buffer space to each priority using a simple static threshold scheme. The exact method of managing buffer space is left up to the implementer.

Xilinx, Motorola, Redswitch, and Praesum are a few of the vendors that offer either core or switch products, and have made available a number of data sheets, whitepapers, and presentations. These documents provide performance figures, but more importantly topical discussions of some of the limitations and challenges of the RapidIO protocol. For our simulations, system parameters such as packet assembly time, packet disassembly time, and input/output queue lengths were taken from these data sheets (refer to [17-25]).

B. *GMTI*

GMTI is an important application in military operations, since moving targets may be laid over a map of a battlefield for strategic planning during a conflict. GMTI works best when combined with some form of airborne radar system. Since space is the ultimate "high ground" for radar systems, GMTI can be very effective in a SBR system. However, one of the main drawbacks of GMTI is that it requires a high degree of processing power, anywhere from 40 to 280 GFLOPs and more, depending upon the size and resolution of the data being collected. This need means that adding GMTI payload processing to a satellite system will further stress the constraints imposed by a limited, embedded architecture in a high-radiation environment.

GMTI is an algorithm composed of several steps, which are Pulse Compression, Doppler Processing, Space-Time Adaptive Processing (STAP), and Constant False-Alarm Rate detection (CFAR). Each phase of the GMTI algorithm depends heavily on signal processing and linear algebra techniques. In order for GMTI data to be useful, it must be processed in real time. A 3-dimensional data cube represents incoming data with each complex number element consisting of a real number of 4 bytes and an imaginary number of 4 bytes. Incoming GMTI data is grouped into Coherent Processing Intervals (CPIs), which are defined as the time interval that contains data that can be processed at a single time by the GMTI algorithm [10]. Due to some of the fundamental properties of signal processing, there is an upper limit to the amount of data that can be coherently processed as a single group. Since data coming in from the radar sensors comes in as a stream, this limit is usually expressed as a time interval, typically 256 milliseconds. This behavior means that every 256 milliseconds a new data set is ready, so this time interval becomes a real-time constraint for the system. Typical values for data cubes range in the size from 10 MBytes for data cubes associated with airborne radar systems to 6 GBytes or more for space-based radar systems due to an increased line of sight for satellite-based systems.

IV. Methodology

In this section, we introduce the parallel partitioning methods for GMTI algorithms considered in this work. We also describe our simulation environment, the models comprising this environment, and their parameters.

A. *GMTI algorithms*

Given the basic GMTI algorithm under consideration, we developed three forms of parallel partitioning: a straightforward partitioning, a staggered approach, and a pipelined approach. We decided to make our partitionings relatively generic so that they can support differing numbers of processing nodes without significant changes needed. In this manner, we could examine the scalability of each approach.

The first partitioning considered was a straightforward mapping of the incoming data set equally across each of the digital signal processor (DSP) nodes. Since the GMTI algorithm is mainly composed of signal processing procedures, and no interprocessor communication is needed during each stage [13], the algorithm can be thought of as embarrassingly parallel. For this partitioning, each processing node performs all four stages of the GMTI algorithm and sends the detection results back to a single destination.

The second partitioning considered was one based off the staggered approach outlined in [13]. This approach is similar to the straightforward mapping, except that the incoming data is sent to different groups of processors in a staggered fashion. In this partitioning, incoming data cubes are sent to groups of processors in a round-robin fashion. For example, given a four-node system, the first two nodes will process the first data cube, the second two nodes will process the second data cube, and so on. In this manner, each processor receives a larger amount of data than it can process at a single CPI, but receives this data less often.

The pipelined GMTI algorithm we are considering is very similar to the algorithm presented in [16]. However, we have adapted the pipelined algorithm presented in that paper to more closely match our target environment. In our version of the pipelined partitioning, we split up the pipeline into four stages, with a pipeline stage for each step of the GMTI algorithm.

In each partitioning, we assume that data comes from a single source and is sinked to the same location, which could possibly involve multi-ported access to a global memory bank. We are only interested in the performance of the algorithm partitioning itself, and so we assume that the global memory bank has sufficient bandwidth (through multiporting or other methods) to handle the incoming and outgoing data bandwidth requirements.

B. *Simulation environment*

Mission-Level Designer (MLD) from MLDesign Technologies Inc. was selected as our primary modeling tool for this project [26]. While MLD is an integrated software package with a diverse set of modeling and simulation domains, the discrete-event domain was used in this study. The tool allows for a hierarchical, dataflow representation of hardware devices and networks with the ability to import finite-state machine diagrams and user-developed C/C++ code as functional primitives. The capability of MLD to model and simulate computer systems and networks is based upon its precursor, the Block-Oriented Network Simulator (BONeS) Designer tool from Cadence Design Systems. BONeS was developed to model and simulate the flow of information represented by bits, packets, messages, or any combination of these. An overview of BONeS can be found in [27].

Three major MLD models were created for use in our simulation: a RapidIO endpoint model, a RapidIO switch model, and a processor model. Our switch models are based around a central-memory switching architecture with time-division multiplexing (TDM). An incoming switch port writes data to the central memory during its TDM slot, while an outgoing port reads data from memory during its TDM slot. We chose to use a packet-level granularity for our models (as opposed to a cycle-level granularity) in order to keep simulation times reasonable. Our models ignore signal propagation delay, because the extremely small distances involved would not have a noticeable impact on the overall results of the simulation. The important parameters supported by our RapidIO-based models are listed below:

- Physical layer clock: Our models support arbitrary clock speeds.
- Input/output buffer sizes: Number of packets that an endpoint can buffer in its incoming and outgoing interfaces.
- Physical layer priority thresholds: These parameters correspond to the levels used in the static priority threshold scheme, which determines how many packets of a certain priority may be accepted based on how much buffer space is currently occupied.
- Physical layer link width: Selects an 8- or 16-bit parallel interface.
- Response assembly time: Delay that is incurred before an endpoint sends a message-passing response to a message-passing request.

Parameters specific to our central-memory switch are listed below:

- Average memory read/write latency: In our switch model, the memory is shared between all ports via TDM. This parameter is used to represent that delay. Since we are not performing cycle-level simulations, the statistically expected value is used for this delay.
- Central memory size: Number of bytes that are available in central memory. Since the switch is a central-memory switch, this memory can be used by any port. We do not restrict single ports from taking over a certain percentage of the central memory; memory is given to each port if it is available. Simple static threshold priorities are used in a similar manner to the static thresholds of the endpoints, and these values are global to each port. For example, one possible static threshold setting for the central memory switch is to only accept the lowest priority packets if there are at least 500 bytes (for instance) of memory available in the switch.
- Cut-through routing: Cut-through routing may be turned on or off in the switch by adjusting this parameter. Cut-through routing eliminates most of the delay of copying a packet to switch memory by sending a packet immediately (subject to the memory read and write latencies) to the next “hop” if no other packets are currently queued to be sent out of the packet's corresponding output port. The alternative to cut-through routing is store-and-forward routing.

We used a generic vector processor to model each processing node in the system and its corresponding DSP functions. In our processor models, incoming packets are buffered until sufficient payload has been received to perform a DSP operation on that data. Once the data has been processed, a certain amount of data is retransmitted to a specifiable number of destinations. In our processor models, we assume asynchronous communication is used, and incoming data is copied to local memory via Direct Memory Access (DMA) so that the processor is not tied up with marshaling incoming data to memory when data arrives. Our generic processor model does not assume pipelined processing; if the processor is currently busy processing one group of data, any other groups of data that are ready to be processed must wait for the processor to finish with its current group before starting processing on any new groups.

C. *Simulation parameters*

For the experiments in this paper, we use an incoming GMTI data set corresponding to a sustained data rate of approximately 4.6 Gbps. In other words, 4.6 Gbps of incoming raw data arrives from the sensors for processing. We started each system with all processors and networks in an idle state, and then sent eight CPIs worth of data to the network, with data being sent out as evenly as possible (streamed) across each CPI.

Our parameters used for the RapidIO components (derived from [28] wherever possible) are listed below:

- Physical layer clock rate: 250 MHz
- Input/output buffer sizes: 8 packets
- Static priority thresholds: 6, 8, 8, 8 packets for priorities 0 (lowest), 1, 2, and 3 (highest), respectively
- Physical layer link width: 16 bits
- Response assembly time: 12 ns
- Packet disassembly delay: 14 ns

The parameters chosen for our switch models are listed below:

- Average memory read/write latency: 72 ns
- Central memory size: 10000 bytes with priority thresholds of 1024, 0, 0, and 0 for priorities 0, 1, 2, and 3, respectively

In our systems, we only have two classes of traffic priorities: priority 0 and priority 1. The message-passing layer always sends responses back with the priority level increased by one, and since all packets that represent the data cube should have the same priority level, we end up with two priority levels. Since we will never have any packets above priority 1, we set the thresholds for priority levels 1, 2, and 3 the same. In our systems, we reserve the last 1024 bytes of the central-memory switch memory and the last 2 spaces in each endpoint's buffer for message-passing responses. In our model, we use the message-passing responses as an application-level acknowledgement that another endpoint has received a data packet.

D. Experiments

Given the parameters listed in the previous section, we built several systems while varying specific parameters. We decided to concentrate on the following variables for our simulations:

- Number of compute nodes: We built systems composed of 8, 12, 16, and 24 processing nodes. Each system was created in three different variants corresponding to one of the three different partitioning strategies. In the straightforward and staggered partitioning strategies, each node performed all four stages of the GMTI algorithm. In the pipelined partitioning strategy, we assigned differing numbers of processors to each subtask based on static performance prediction that took place before the simulation was run. Since the amount of work for each stage can be accurately predicted beforehand, we felt a dynamic assignment of processors to tasks during runtime would introduce a significant amount of overhead with little or no benefit. For each system, we tailored a specialized interconnect that was geared towards the pipelined algorithm, since that has the most communication requirements. To ensure we were not unfairly favoring the pipelined algorithm, we built a 16-node system with an interconnect specifically designed for the straightforward and staggered partitioning approaches.
- Cut-through on/off: For each of the systems listed above, we conducted experiments with cut-through routing turned both on and off on the RapidIO switching fabric.
- Flow-control method: For each of the systems listed above, we conducted experiments that used both transmitter-controlled and receiver-controlled flow control for the RapidIO fabric.

In each experiment, a total of eight CPIs of data were sent to the processors and the time taken to complete each iteration was recorded. In addition, statistics on the average packet delays, total amount of data transferred, and computation time per processor were recorded.

V. Results

In this section, we present the results obtained from running our simulation experiments. Additional analysis of these results will be presented in the following section.

One of the first things we were interested in was whether the two improvements to the RapidIO fabric we considered, cut-through routing and the flow-control method, made a measurable impact on the overall performance of the system. After analyzing each of the systems, we have concluded that these two improvements do not make a marked improvement on the performance of the system for the GMTI algorithm under study. Cut-through routing definitely helped reduce average packet latencies, but did not significantly impact the amount of time taken to compute a data cube or the total throughput in the system (the differences in times were less than 10 microseconds, and most of the results had times in the millisecond to second range). Using transmitter-controlled flow control did not make any measurable impact at all for most of the systems, and the systems where it did make an impact were affected in a very minute way. In the interest of conciseness, we will leave out the results from the flow-control simulations. In addition, our interconnect specifically tailored for the straightforward partitioning of the 16-node system did not perform significantly differently than the "regular" architecture so we omit those results in the interest of conciseness as well.

Our primary focus during these experiments was to examine the specifics of how the RapidIO interconnection network performed under each system configuration. Figure 1 shows system-level and application-level bandwidth consumed by each configuration. We define the system-level bandwidth reported in the graph as the total number of bytes transferred in the system divided by the time taken to compute the eight data cubes. The application-level bandwidth shown is the total number of bytes of actual data payload transferred divided by the same total time. For all system sizes, the pipelined partitioning method consumed the most system bandwidth, followed by the straightforward partitioning. The staggered approach resulted in the least amount of network bandwidth being consumed, especially as system size increased. Of course, the bandwidth levels provided by these systems are many times higher than those possible with a bus-based system.

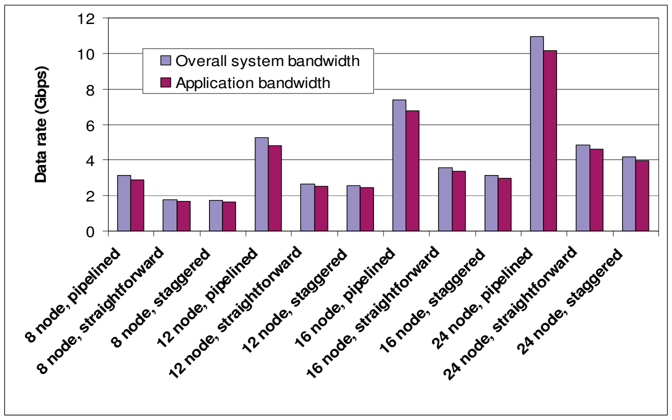

Figure 1: Overall system and application bandwidth

Figure 2 presents the overall packet overhead efficiency, which we define to be the amount of payload transferred in the system divided by the actual number of bytes that were transferred. From this figure, we see that the pipelined partitioning method has the worst packet overhead efficiency, and the other two partitioning methods perform similarly. The reason for the reduced efficiency of the pipelined method in these cases is because the data grouping of this method requires many packets to be sent that contain less than the RapidIO maximum payload size of 256 bytes. Because this data is not necessarily aligned to a valid RapidIO packet size, the packets must be padded with "dummy" bytes, thus consuming system bandwidth and reducing communication efficiency. In our last figure related to communication across the RapidIO fabric, the average packet delays are summarized in Figure 3. Considering the heavy traffic conditions on the network, we find that the RapidIO network provides excellent performance, with all average latency values between 3 and 7 microseconds.

Finally, in an effort to determine which of the partitionings result in systems that can operate in a real-time environment, we recorded the overall throughput (in terms of data cubes processed per second) and compare that to the minimum value needed for real-time processing of data. The resulting data is summarized in Figure 4. The horizontal line in the figure represents the needed throughput rate in order for the system to keep up with the incoming data in real time. In order to keep our simulations as accurate as possible, we did not send out data faster than our defined image sensors could provide. Therefore, since the incoming data is entering the system at a fixed rate, it is not possible for the system to have greater throughput than the line indicated in the figure.

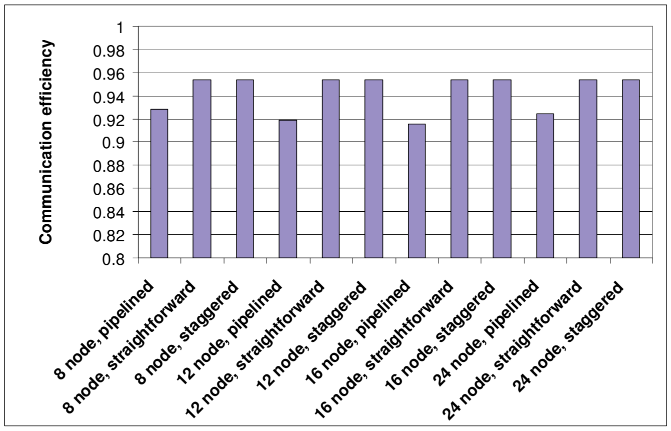

Figure 2: Overall packet overhead efficiency

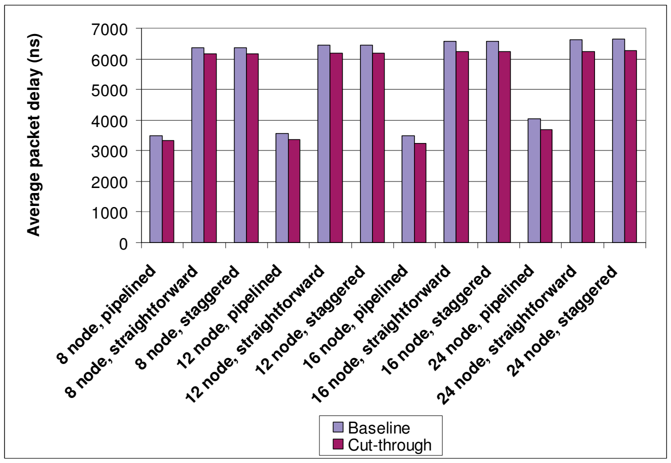

Figure 3: Average packet delay

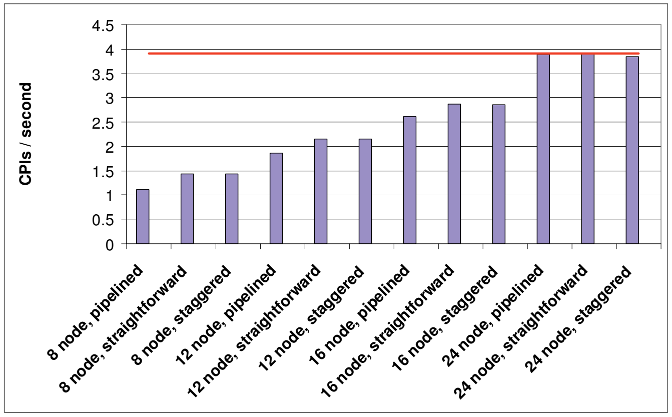

Figure 4: System throughput

VI. Analysis

Several interesting results were obtained from the simulation results given in the previous section. In this section, we will analyze the results and provide some insight into the behavior of our simulations.

A. *Cut-through routing*

At first, we were somewhat surprised that enabling cut-through routing on the switching fabric did not make a noticeable impact on the performance of each of the systems.

However, if we re-examine our algorithm and network architecture, the reason becomes clear. Cut-through routing does provide reduced packet latencies (as shown in Figure 3), but the GMTI algorithm under study is not extremely sensitive to these packet latencies. Throughput is far more important to the GMTI algorithm, and cut-through routing does not provide any additional network throughput.

B. *Transmitter-controlled flow control*

Simulation results showed that transmitter-controlled flow control does not significantly affect the overall system performance with our application (in most cases, especially with the smaller systems, the overall performance stayed exactly the same). With transmitter-controlled flow control, no extra buffer space is added, and the overall clock rate stays the same. In other words, if an endpoint is full and cannot accept another packet, no protocol modification can change that fact. However, it is clear that transmitter-controlled flow control has the potential to be much more energy-efficient than the default receiver-controlled flow control. In basic tests we have performed, transmitter-controlled flow control reduced the number of unnecessary packet transmissions greatly when end nodes were approaching saturation. Because of this fact, and because we did not notice any performance degradation associated by using transmitter-controlled flow control instead of receiver-controlled flow control, we recommend its use in environments where power consumption needs to be minimized.

C. *Analysis of each partitioning method*

By examining the results presented in the last section, it is clear that the straightforward partitioning method seems to be the most likely candidate for parallelizing the GMTI algorithm for SBR. The pipelined approach is unable to sustain sufficient throughput to keep up with a real-time system, even with 24 nodes, and the staggered approach did not seem to improve system throughput while incurring slightly increasing latencies and having a worse parallel efficiency than the straightforward decomposition. It would be easy to dismiss both the staggered and pipelined partitioning methods, but we have decided to examine them in detail to determine what may be the cause of their suboptimal performances.

We first examine the staggered processing approach. An interesting result was observed in comparing the processor utilizations of the staggered partitioning compared to other algorithm partitionings. We observed that the processors in each system seem to be incurring more idle time as the size of the system increases. However, if we examine Figure 4 more closely, in every system the throughput of the staggered approach is less than the throughput of the straightforward approach. If the processors were idle because they were processing their data faster than the data was coming into the system, this would not be the case. Therefore, there must be another reason for this increase in processor idle time. We have two explanations for this behavior:

1. At the beginning of the simulation, at least one group of processors will always sit idle until the data source sends out enough data cubes for these processors to begin working. In our 16- and 24-node systems, we used four groups of processors instead of two groups, and this is reflected by the increase in idle time in these larger systems.
2. In our switch topologies, groups of processors are organized to be close together. Because of this, the average packet latency to each processor group will either be very good or very bad. We notice that even though the overall packet latencies stay manageable (as shown in Figure 3), having a non-uniform packet delay may penalize the groups of processors that are farthest away from the data source. If the delay is large enough due to contention and other issues, a processor may finish its work and remain idle until new work arrives. Even though there is more than enough work for that processor to do, it may be "stuck" in the interconnection network. Note that in the straightforward approach, this trait is less likely since traffic is fairly evenly spread out across the interconnection network.

Redesigning our interconnection networks specifically for the staggered processing method may mitigate the second item mentioned above. However, the first item mentioned above is a fundamental limitation in the staggered processing approach. It is for these reasons (i.e. unnecessary idling of processors, sensitivity to interconnection topology) that we do not recommend the use of the staggered processing algorithm. It is interesting to note that staggered processing was effectively used in [13] on an Intel Paragon supercomputer. However, in that case, the processing was performed under much different conditions. Relatively speaking, the ratio of network speed to the data sizes used in that experiment is much higher for the Paragon as compared with the systems we are considering. This difference raises an interesting point: many strategies that are successful for conventional off-line parallel computing fail in high-performance embedded parallel computing because of size, weight, or power constraints.

We now consider the pipelined approach. One obvious drawback to this approach is the increased amount of communication necessary to process an incoming data cube. Since this increase in communication cost does not yield any benefits in system throughput, the value of the pipelined approach may clearly be questioned, especially if generic compute nodes are used in each pipeline stage. However, consider a traditional processor pipeline for a moment. The power of pipelining comes by stringing together smaller, less complicated stages to increase system throughput. If generic compute nodes are used that have the same cost as a compute node, and these nodes can perform all stages of the GMTI algorithm without any synchronization, then it does not make much sense to trade in more communication overhead for no improvements in throughput and increases in system latencies. However, if each compute node can be constructed such that it is specialized to its particular task, and this can be done at a lower cost than using a generic compute node, then a pipelined approach can offer a similar performance at a reduced cost. For example, if each compute node in the 24-node pipelined system costs half as much or consumes half as much power as the straightforward 24-node approach, then the 24-node pipelined system can be equally compared to the 12-node straightforward approach. Under this assumption, the 24-node pipelined system clearly outperforms the 24-node straightforward approach.

One additional benefit (which is hinted by the results from Figure 3) is that pipelining can actually reduce communication costs in systems where costly all-to-all communication is needed. Since pipelining can decrease the number of processors that need to participate in the all-to-all communication event, it may be possible to group these nodes closer together on the interconnect fabric, and so the cost needed to perform the all-to-all communication (and any other costly group communication) can be drastically reduced. This benefit may more significant for other space-based radar applications (such as synthetic aperture radar or SAR) that do require group communication between processors.

VII. Conclusions

In this paper, we have presented research on RapidIO and the GMTI space-based radar algorithm. We have proposed three different partitioning methods of GMTI that can be performed over a RapidIO interconnection network and simulated each of these partitioning methods under a variety of conditions. In general, we found that many techniques useful in conventional HPC environments are less effective under the size, weight, and power constraints imposed in an embedded system. For our application, simulation results have shown that adding cut-through routing to the switching fabric does not result in a noticeable improvement in performance. Results also indicated that transmitter-controlled flow control method does not improve performance, but it does present an opportunity to reduce power consumption under certain conditions with no negative impact in performance. We have determined that, given our assumptions of the GMTI algorithm, a straightforward partitioning of this algorithm provides the best performance on a RapidIO system, while the pipelining partitioning method may provide an excellent option for reducing total system cost. Overall, our experiments showed that RapidIO is a promising and viable platform for GMTI, with capabilities far exceeding those of traditional bus-based systems. With network efficiencies greater than 90% and average packet latencies less than 7 microseconds, the RapidIO network performed well even under the heavy network load required by the real-time GMTI algorithm.

References

[1] G. F. Pfister, "An introduction to the InfiniBand architecture," in *High Performance mass Storage and Parallel I/O: Technologies and Applications* (H. Jin, T. Cortes, and R. Buyya, eds.), ch. 42, pp. 617-632, New York, NY: IEEE Computer Society Press and Wiley, 2001.

[2] D. Gustavson and Q. Li, "The Scalable Coherent Interface (SCI)," in *IEEE Communications*, pp. 52-63, Vol. 34, No. 8, August 1996.

[3] "RapidIO Interconnect Specification Documentation Overview," RapidIO Trade Association, June 2002.

[4] "RapidIO Interconnect Specification (Parts I-IV)," RapidIO Trade Association, June 2002.

[5] "RapidIO Interconnect Specification, Part V: Globally Shared Memory Logical Specification," RapidIO Trade Association, June 2002.

[6] "RapidIO Interconnect Specification, Part VI: Physical Layer 1x/4x LP-Serial Specification," RapidIO Trade Association, June 2002.

[7] "RapidIO Interconnect Specification, Part VIII: Error Management Extensions Specification," RapidIO Trade Association, September 2002.

[8] "RapidIO Hardware Inter-operability Platform (HIP) Document," RapidIO Trade Association, November 2002.

[9] T. Nohara, P. Weber, A. Premji, and C. Livingstone, "SAR-GMTI processing with Canada's Radarsat 2 satellite," in *IEEE Adaptive Systems for Signal Processing, Communications, and Control Symposium*, pp. 376-384, 2000.

[10] T. L. Hacker, "Performance Analysis of a Space-based GMTI Radar System using Separated Spacecraft Interferometry," Master's thesis, Massachusetts Institute of Technology, May 2000.

[11] M. Skalabrin and T. Einstein, "STAP Processing on Multiprocessor Systems: Distribution of 3-D Data Sets and Processor Allocation for Efficient Interprocessor Communication," in *Proceedings of the Adaptive Sensor Array Processing (ASAP) Workshop*, MIT Lincoln Laboratory, Lexington, MA, March 1996.

[12] M. Lee and V. K. Prassana, "High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing," in 2nd *International Workshop on Embedded Systems and Applications*, Geneva, Switzerland, April 1997.

[13] R. Brown and R. Linderman, "Algorithm Development for an Airborne Real-Time STAP Demonstration," in *IEEE National Radar Conference*, Syracuse, NY, 1997.

[14] M. Linderman and R. Linderman, "Real-Time STAP Demonstration on an Embedded High Performance Computer," in *IEEE National Radar Conference*, Syracuse, NY, May 1997.

[15] A. Choudhary, W. K. Liao, P. Varshney, D. Weiner, R. Linderman, and M. Linderman, "Design, Implementation, and Evaluation of Parallel Pipelined STAP on Parallel Computers," tech rep., Northwestern University, 1998.

[16] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, M. Linderman, and R. Brown, "Design, Implementation, and Evaluation of Parallel Pipelined STAP on Parallel Computers," in *IEEE 12th International Parallel Processing Symposium*, Orlando, FL, pp. 220-225, March 1998.

[17] Redswitch, "RS-1001 16-Port, Serial/Parallel RapidIO Switch and PCI Bridge," Product brief, 2002.

[18] T. S. Corporation, "Tsi500 Parallel RapidIO Multi-port Switch," Feature sheet, 2003.

[19] T. S. Corporation, *Tsi500 Parallel RapidIO Multi-port Switch User Manual*, 2003.

[20] Xilinx, "LogiCORE RapidIO 8-bit Port Physical Layer Interface Design Guide," Xilinx Intellectual Property Solutions, 2003.

[21] Xilinx, "Xilinx Solutions for RapidIO," Presentation, June 2002.

[22] Xilinx, "Xilinx LogiCORE RapidIO Logical (I/O) and Transport Layer Interface, DS242 (v1.3)," Product Specification, 2003.

[23] Xilinx, "Xilinx LogiCORE RapidIO 8-bit Port Physical Layer Interface, DS243 (v1.3)," Product Specification, 2003.

[24] L. Logic, "Leopard Logic unveils RapidIO implementation platform based on its Hyperblox embedded FPGA cores," Press release, July 2002.

[25] P. Communications, "RapidIO Physical Layer 8/16 LP-LVDS Core Product Brief," Product brief, 2001.

[26] G. Schocht, I. Troxel, K. Farhangian, P. Unger, D. Zinn, C. Mick, A. George, and H. Salzwedel, "System-Level Simulation Modeling with MLDesigner," in 11th *IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Orlando, FL, October 2003.

[27] V. F. K. Shanmugen and W. LaRue, "A Block-Oriented Network Simulator (BONeS)," Simulation, Vol. 59, No. 2, February 1992.

[28] Motorola Semiconductor Product Sector, "RapidIO: An Embedded System Component Network Architecture," RapidIO Whitepaper, February 2000.